Preprint
UCRL-ID-148317

Java Performance for Scientific Applications on LLNL Computer Systems

C. Kapfer, A. Wissink

U.S. Department of Energy

Lawrence
Livermore
National
Laboratory

May 10, 2002

Java Performance for Scientific Applications on LLNL Computer Systems[*]

Craig Kapfer, Andrew Wissink
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
{ckapfer,awissink}@llnl.gov

1 Overview

Languages in use for high performance computing at the laboratory - Fortran (f77 and f90), C, and C++ - have many years of development behind them and are generally considered the fastest available. However, Fortran and C do not readily extend to object-oriented programming models, limiting their capability for very complex simulation software. C++ facilitates object-oriented programming but is a very complex and error-prone language. Java offers a number of capabilities that these other languages do not. For instance it implements cleaner (i.e., easier to use and less prone to errors) object-oriented models than C++. It also offers networking and security as part of the language standard, and cross-platform executables that make it architecture neutral, to name a few. These features have made Java very popular for industrial computing applications. The aim of this paper is to explain the trade-offs in using Java for large-scale scientific applications at LLNL.

Despite its advantages, the computational science community has been reluctant to write large-scale computationally intensive applications in Java due to concerns over its poor performance. However, considerable progress has been made over the last several years. The Java Grande Forum [1] has been promoting the use of Java for large-scale computing. Members have introduced efficient array libraries, developed fast just-in-time (JIT) compilers, and built links to existing packages used in high performance parallel computing. For example, mpiJava [2] is an interface between Java and the MPI message passing library. In addition, at the ACM 2001 Java Grande/ISCOPE Conference [3], a number of researchers reported that their Java codes closely matched or even *exceeded* the performance of comparable C and Fortran implementations. The results primarily focused on the timing of serial kernels and often used proprietary (i.e., not publicly available) implementations of Java compilers. A comprehensive analysis of Java for use in large-scale scientific applications, i.e., an analysis of the memory usage, performance with publicly available compilers, performance on distributed memory parallel computers, and overall application performance (as opposed to kernel performance), was not discussed.

This study investigates performance of scientific applications written in Java on a number of different systems at LLNL. The first section of this report discusses several benchmark applications and compares memory and run-time performance of the Java

[*]This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract number W-7405-Eng-48.
UCRL-ID-148317.

benchmarks (run sequentially) to comparable applications written in C. The next section discusses parallel performance of Java using mpiJava [2] on several systems including a Sun cluster of workstations and the IBM Blue-Pacific parallel system at LLNL and provides a performance comparison of mpiJava to standard MPI. The last section discusses the parallel performance of two scientific applications that use mpiJava. We conclude with remarks on our findings and recommendations for future work.

2 Benchmarks

The Edinburgh Parallel Computing Center (EPCC) [4] at the University of Edinburgh has developed a benchmark suite of applications [5] aimed at testing aspects of Java execution pertinent to large-scale scientific applications. The benchmarks consist of low-level operations, such as loop overheads and various math functions, kernels such as heap-sort, Fast Fourier Transforms, etc., and so-called large-scale applications like computational fluid dynamics, molecular dynamics simulations, etc. These benchmarks are written in Java and come with comparable codes written in C by which we can compare performance. Due to the limited time to complete this study, we investigated performance only of the large-scale application suite.

The applications we investigated were chosen for their similarity to applications codes used at LLNL. Specifically, we selected an Euler computational fluid dynamics application, a molecular dynamics application, and a ray tracer graphics-based application.

The Euler application solves time-dependent Euler equations for flow in a channel with a bump on one of the walls. It uses a structured irregular N x 4N mesh, and the solution method is a finite volume scheme using fourth order Runge-Kutta with 2nd and 4th order damping. The solution is iterated for 50 steps. It was written by David Y. Oh of the Computational Aerosciences Laboratory at MIT. A web-based simulation is available online at `http://raphael.mit.edu/Java/`.

The Molecular Dynamics application is an N-body code which models particles interacting under a Lennard-Jones potential in a cubic spatial volume with periodic boundary conditions. The original Fortran code was written by Dieter Heerman, Institut fur Theoreische Physik and converted to Java by Lorna Smith, EPCC.

The ray tracer application renders a scene with 64 spheres. The resolution of the render is N x N pixels.

Different sizes were chosen, as shown in table 1. The size of some of the benchmarks was manually extended to increase the problem size.

Problem	**Case A**	**Case B**	**Case C**	**Case D**	**Case E**
Euler - Gridcells	16.7K	37.3K	148.4K	331.8K	475.2K
Molecular Dynamics - Particles	2048	8788	–	–	–
Ray Tracer - Pixels	22.5K	250K	–	–	–

Table 1
Problem sizes of Java benchmark calculations.

3 Serial Performance

The Euler Gas Dynamics and Molecular Dynamics applications discussed in section 2 have implementations in both Java and C. The algorithms and problem conditions for the two languages are identical. The difference between the Java and C implementations is that

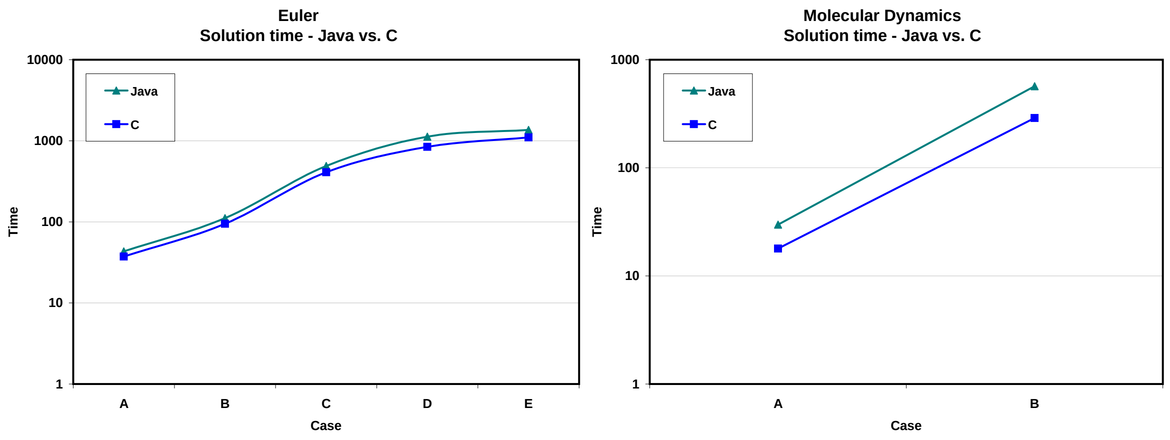


FIG. 1. *Serial runtime for Java and C versions of Euler and Molecular Dynamics JGF benchmarks on Sun Ultra 100.*

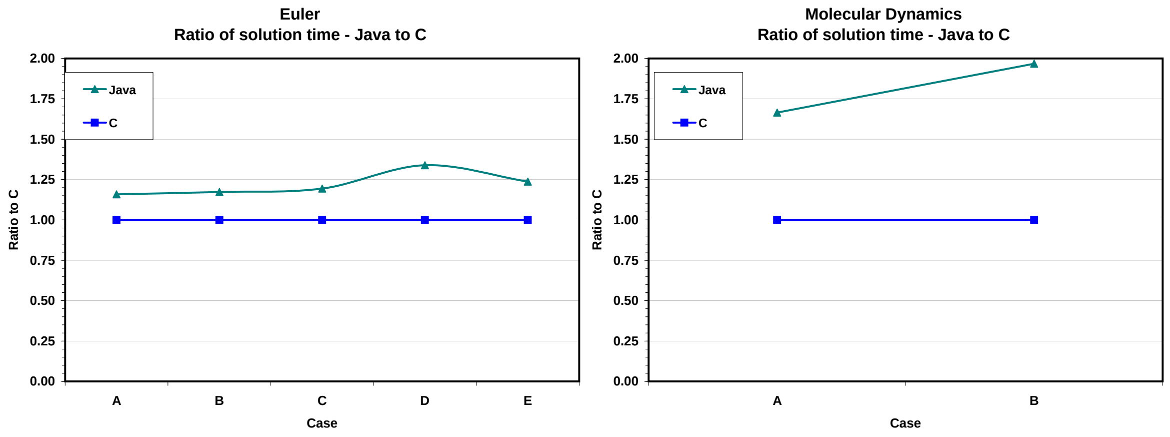


FIG. 2. *Ratio of Java and C serial runtimes for Euler and Molecular Dynamics JGF benchmarks on Sun Ultra 100.*

the Java version uses some high level objects (as in an object-oriented implementation) whereas the C version uses a purely structured coding style. We compare the performance of the Java and C versions of these applications, both in solution time and in memory, on several computer systems at LLNL.

Figures 1a and 1b show the measured run-times of the C and Java versions of the Euler and Molecular Dynamics applications using various problem sizes run on a Sun Ultra 80. This machine has a single 440MHZ Sparc processor and 4 GB RAM. The machine was essentially dedicated as there were no other significant-sized jobs running while these timings were performed. The different cases represent different problem sizes, as discussed in section 2. The Sun Workshop Pro C compiler, cc, with optimization level 3 (i.e., -O3) was used to compile the C cases. The standard javac compiler and JVM interpreter included with the Java v1.2 release were used for the Java code. Figures 2a and 2b show the ratio of the solution time using Java to the time using C.

The run-times for the Java version of the Euler application were comparable to those for the C version. The ratio of Java to C was consistently between 1.2 to 1.25 for the different problem sizes tested, although case D shows slight deviation with a ratio of 1.35.

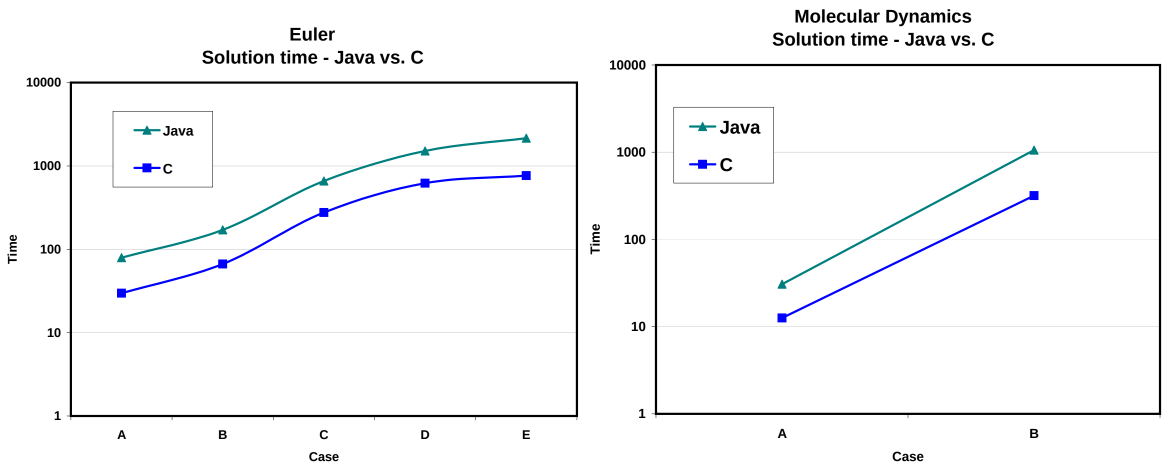


FIG. 3. *Serial performance for Java and C versions of Euler and Molecular Dynamics JGF benchmarks on IBM ASCI Blue-Pacific.*

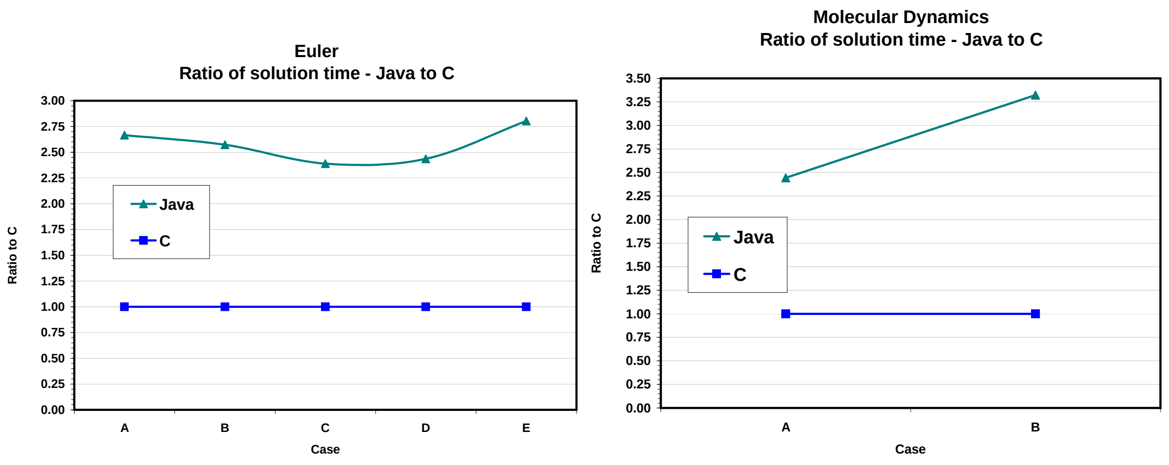


FIG. 4. *Ratio of serial benchmark performance between Java and C on IBM ASCI Blue-Pacific.*

The run-times for the Molecular Dynamics code show a greater disparity. The ratio between the Java and C run-times for this case varied from 1.7 to 2.0. We postulate that the performance degradation for the Java version of this application, relative to C, results from a higher number of memory fetches. The Molecular Dynamics code uses much less memory than the Euler case and does many more numerical operations on a particular piece of data once it has been accessed from memory. If the compiler is smart enough to recognize this behavior, it can optimize the code to assure that often-used data is maintained in cache rather than accessing it from main memory. We do not expect the Java compiler to be as good as its C counterpart in recognizing such constructs since it is not generally used for performance-based applications.

Figures 3a and 3b, and figures 4a and 4b, show results for the same two problems run on a single processor of an SMP node on IBM ASCI Blue-Pacific. Each node of Blue contains four 332MHZ PowerPC 604e processors with access to 1.5 GB RAM. The memory is shared between all 4 processors on a node. IBM's xlc compiler with optimization level 3 was used to compile the C cases. The standard javac compiler included with the Java 1.3 release was used for the Java code. The ratio of Java to C for the Euler application run on a single processor of Blue varied between 2.3 to 2.8 for the various problem sizes tested. For the molecular dynamics case, the ratio varied between 2.4 and 3.3.

The run-time ratios between Java and C are significantly larger on the IBM than on the Sun. We postulate that the standard Java compiler and JVM are less optimized for the IBM SP system than they are for Sparc systems. This stands to reason since Sun releases Java and developers are likely more focused on Sun systems. It should be noted, however, that IBM has released optimized just-in-time (JIT) JVMs for Java which claim to have considerably better performance than the standard JVMs. Because these are only available for evaluation and because root access is generally required to install them, we were unable to evaluate their performance.

3.1 Memory Usage

Memory is a primary computational limitation for many large-scale scientific applications. While we found a number of studies in the literature that addressed the *speed* of Java relative to other languages, we did not find any memory usage comparisons. This motivated us to investigate memory usage characteristics of Java relative to C.

We collected memory usage statistics for Java and C implementations of the Euler code on the Sun Ultra 80 and IBM Blue-Pacific. After some analysis, we determined there were four different measures of memory to consider during the study, listed here in order of decreasing size:

1: *The total memory, which includes the memory requested by the JVM to execute the Java byte-code together with the overhead for the JVM itself.*

2: *The block of memory that the JVM reserves to allow the application to have access to (reported by Java internals).*

3: *The memory resident within the program at a given time.*

4: *The amount of memory the application actually uses inside the JVM.*

The total memory and the resident memory (i.e., 1 and 3) are measured using by the Unix *top* utility. The total memory is the reported total "size" of a process while the resident memory is the reported "resident" memory of the process. The block of memory reserved by the JVM and the amount actually used by the application (i.e., 2 and 4) are reported by direct calls to Java internals (e.g., Runtime.getRuntime().totalMemory()).

	Resident	**Size**	**JVM Overhead**
Java Loop	11.0 MB	31.0 MB	20.0 MB
Java Sleep	7.5 MB	31.0 MB	23.5 MB
C Loop	672 KB	912 KB	not applicable
C Sleep	736 KB	912 KB	not applicable

Table 2

Memory Usage Results for "Hello World" programs reported by top.

3.1.1 Hello World

We determined early-on that the JVM interpreter that runs the compiled Java executable invokes a certain amount of memory overhead. In order to characterize this amount, we constructed two "Hello world" programs in both C and Java and tested them on the Sun Ultra 80. The first program contained a loop around a print statement so that the program would run long enough to measure memory statistics. The second contained a system call to sleep(). Table 2 reports the memory usage statistics using *top* (i.e., 1 and 3 in the list described above) while table 3 reports memory statistics as reported by Java internals (i.e., 2 and 4 in the above list).

	Total	**Free**	**In Use**
Java Sleep	8.0 MB	7.8 MB	133 KB
Java Loop	8.0 MB	7.5 MB	437 KB

Table 3

Memory Usage results for Java "Hello World" program reported by Java internals.

The total size of both Java "Hello World" programs reported by *top* was 31 MB. Of this 31 MB, only 8 MB was available to the program, but Java internal routines reported it was actively using only 133 KB. Similar C programs required less than 1 MB (by all measures) of memory. Interestingly, *top* reports the resident memory usage for the two Java programs differed by 3.5 MB while the memory usage reported by Java internals varied only by 300 KB.

We infer from these statistics that there is a general memory overhead associated with the JVM. We also noticed that the program has a segment of memory allocated for it to use which is larger than it actually needs. We defined this overhead to be the difference between the total memory reported by *top* and the total memory available to the program as reported by Java internals. The JVM memory overhead ranged from 20-23.5 MB for the "Hello World" programs (see table 2).

3.1.2 Euler As the Euler code was the most memory intensive application of the benchmarks provided in the suite, we report memory usage statistics for it on the Sun Ultra 80 and ASCI Blue-Pacific (see table 4 and figure 5). On the Blue-Pacific system, we could not run *top* as we do not have login access to the nodes that the program actually runs on. However, there is another utility on Blue, *jr*, which allows us to collect memory statistics for each node of a given job and we used those statistics rather than *top*'s. We believe the output of the *jr* utility corresponds with *top*'s resident memory statistic although documentation on *jr* [6] did not clarify that.

Euler	**C - *top***	**Java - *internals***	**Java - *top***
Sun Ultra 80			
Case A	6.8 MB	6.3 MB	19.0 MB
Case B	13.0 MB	13.7 MB	28.0 MB
Case C	50.0 MB	53.3 MB	76.0 MB
Case D	110 MB	119 MB	152 MB
Case E	135 MB	146 MB	188 MB
IBM ASCI Blue-Pacific			
Case A	6.1 MB	9.8 MB	20.1 MB
Case B	13.0 MB	21.1 MB	35.2 MB
Case C	50.2 MB	81.9 MB	123.1 MB
Case D	112 MB	183 MB	251 MB
Case E	138 MB	225 MB	271 MB

Table 4

Comparison of Java and C Memory Usage results for the Euler benchmark application.

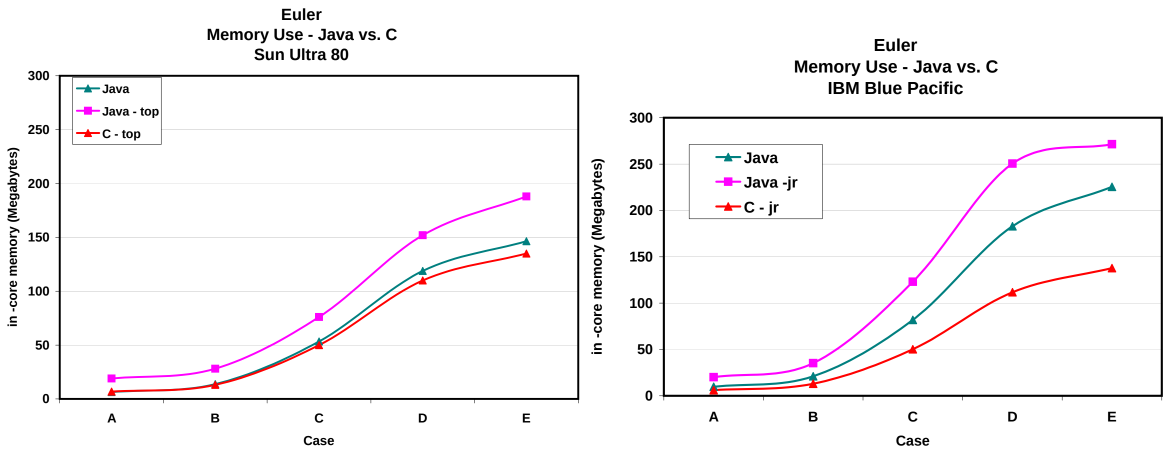


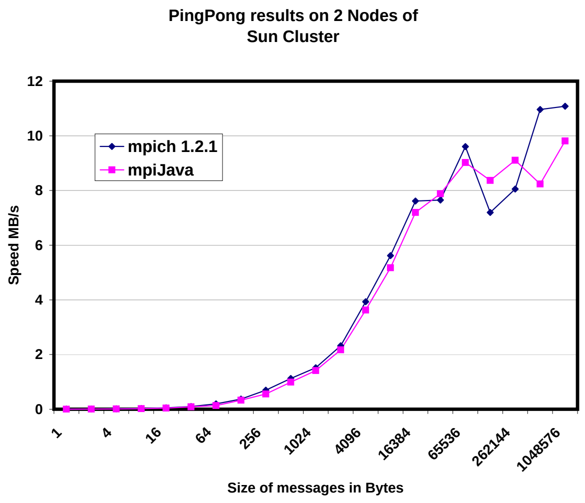
Fig. 5. *Memory usage of Euler benchmarks on Sun Ultra 80.*

Five different problem sizes were tested. Table 4 and figure 5 show the total memory as reported by top for C and Java implementations (labeled “C-top” and “Java-top”, respectively) and internal memory used by the Java application as reported by Java internals (labeled “Java”). We characterize the JVM overhead as the difference between the memory used by the Java application reported by Java internals and the total memory of the Java process reported by *top*.

On the Sun, the internal memory used by the Java application is comparable to C. The internal memory used by the Java version is about 8% larger than C for the largest case. The JVM overhead varies from 13MB for the smallest case, which is comparable to that observed for the “Hello World” cases, to 42MB for the largest case. We infer from these results that the JVM overhead can significantly add to the memory requirements for smaller-scale Java applications but that it constitutes a proportionately smaller percentage for larger scale applications. On the largest case tested, the Java implementation required 40% more total memory than the C implementation, mainly due to the JVM overhead.

On Blue-Pacific, the variation of the overhead with problem was more pronounced. In case E (the largest), the Java internal memory on the Sun is only 11 MB more than the memory used by the C version, but it is 87MB more on the IBM. The overhead of the JVM also seemed to grow with the problem size. The reasons for this are unclear and we were unable to characterize the nature of this relationship. One reason could be the quality of the JVMs. In the serial run-time results, we found Java code ran slower, proportional to C, on the IBM than it did on the Sun. This led us to suspect that the standard JVM is less optimized for the IBM SP system than they are for Sparc systems. The same could be true of memory management within the JVM. The results may differ if we were to use one of the more advanced proprietary JIT-enabled JVMs from IBM. Further study of how Java allocates and manages memory would be useful to address this issue in future studies.

In spite of the relatively poor memory results we find for Java on Blue-Pacific, we are encouraged by the Sun results because they indicate that with a proper JVM, the memory requirements of Java applications may be comparable to that of C applications. Of course, there will always be some overhead associated with Java due to the JVM. While this overhead can be significant with small-scale applications, it constitutes a smaller percentage of the overall memory used for large-scale applications that are of interest to LLNL.



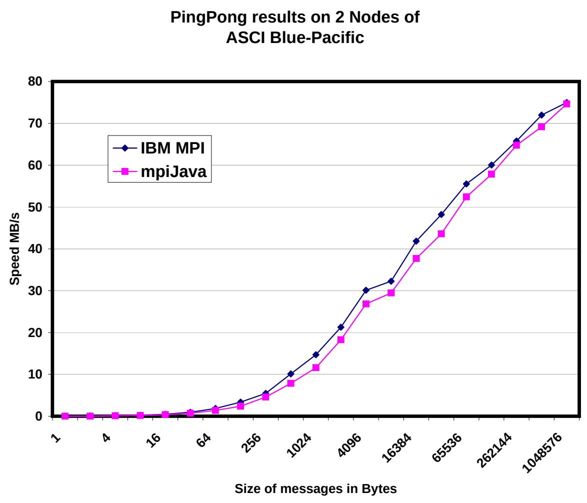


Fig. 6. *Performance of MPICH and mpiJava on Sun cluster; Performance of IBM's MPI and mpiJava on 2 nodes of ASCI Blue-Pacific.*

4 Parallel Performance

This section documents the parallel performance of Java using mpiJava. We tested parallel performance in two separate ways. First we tested the performance of mpiJava in and of itself by running several benchmarks provided with mpiJava on systems at LLNL to investigate the costs of performing communication through the mpiJava interface. Second, we ran several of the parallel applications benchmarks provided by the EPCC in their benchmarks suite [5] on larger scale parallel systems to investigate scaling qualities of Java applications.

The mpiJava package provides an object-oriented Java interface to the MPI standard, for use on parallel or distributed computing platforms. The release includes Java Native Interface (JNI) C stubs that bind the Java interface to an underlying native MPI C interface such as MPICH. Also included is a comprehensive test suite for the Java interface, created by translating the IBM MPI test suite to Java, along with some simple examples and demos [7].

4.1 mpiJava Performance

We tested mpiJava on systems at LLNL by running several of the mpiJava benchmarks. One of the examples included had implementations in both C and Java. This "PingPong" program sends increasing sized messages between processes and measures bandwidth as well as latency of MPI sends and receives. In this context, it serves as a way of measuring overhead associated with Java's calls to the native MPI libraries. We found no appreciable overhead in using mpiJava as compared to C/MPICH (see figure 6) on the cluster of CASC workstations or as compared to the C/IBM MPI on ASCI Blue-Pacific[1]. In fact, during some of the runs on both systems, the mpiJava version performed slightly better than the C version–we attribute these differences (both positive and negative) to system noise.

[1]We tested both inter and intra-node performance ASCI Blue-Pacific. There was little difference in performance between the C and the Java version of the PingPong program in either case. Figure 6 includes the inter-node results.

4.2 Parallel Benchmark Application Performance

Several of the application codes provided in the EPCC's benchmark suite contained parallel implementations (in Java only), including the Molecular Dynamics application discussed in section 3. Unfortunately, the Euler case we tested in serial did not contain a parallel implementation, so we had to substitute another parallel benchmark. We chose to test the parallel implementation of the Ray Tracer benchmark. Ray tracing is a common algorithm used in scientific visualization.

Figures 7a and 7b show the parallel performance of the the Molecular Dynamics and Ray Tracer applications on IBM ASCI Blue-Pacific. Two problems, of increasing size (denoted case A and case B) were run. Both cases are are fixed in size on all processors so traditional speed-up is being measured. It is clear that both algorithms show some degree of scalability. As expected, better scaling is observed for the larger sized problems.

Unfortunately, none of the C versions of the benchmarks were implemented in parallel so it is difficult to compare the parallel performance of Java and C. However, based on results of the serial and mpiJava tests conducted so far, we postulate that the Java cases would probably show similar degrees of scaling to C and would be about 2 to 3 times slower overall.

The main conclusion drawn from parallel tests of these benchmarks is that it is possible to run reasonably large-scale scientific applications written in Java on ASCI-class systems.

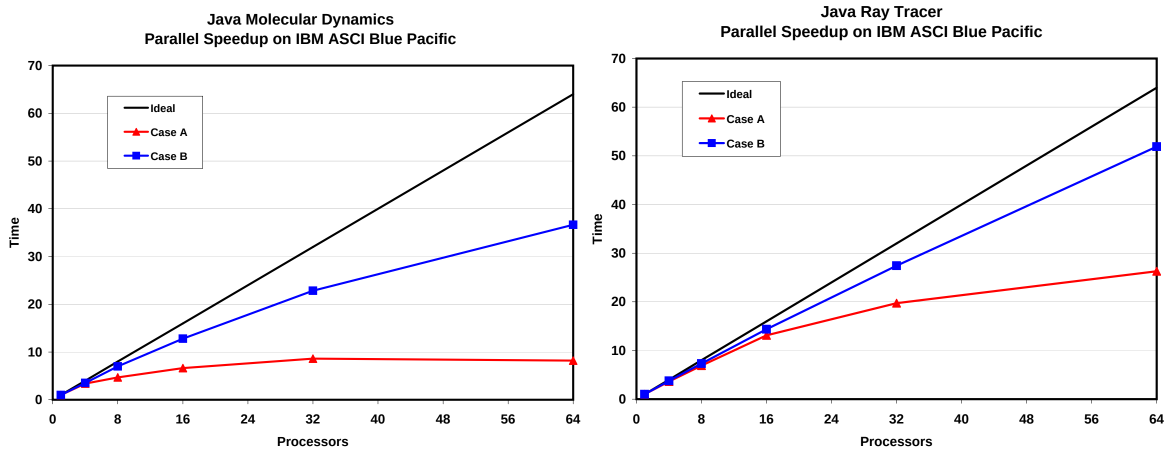


FIG. 7. *Parallel performance of Molecular Dynamics and Ray Tracer benchmarks on ASCI IBM Blue-Pacific. Cases A and B represent different sized problems, with case B the larger of the two.*

5 Parallel Performance

5.1 Caveats, Gotchas, etc.

This section describes some of the difficulties this study presented.

We attempted to use mpiJava on the Tera Cluster 2000 (TC2K) system at LLNL. Unlike the Linux and Sun clusters which use MPICH, TC2K uses a proprietary version of MPI from Compaq that is optimized for the inter-node communications that occur on TC2K. The proprietary version contains an identical API to MPICH, so applications which run with MPICH generally run with the proprietary MPI by simply linking with a different set of libraries. However, mpiJava is more tightly coupled with the MPICH libraries and we ran into some problems when trying to link with the proprietary version. Hence, we were unable to get mpiJava to work on TC2K and, consequently, could not do any parallel tests on the machine. It would be useful to collaborate with the maintainer and developer of mpiJava, Bryan Carpenter, in order to use mpiJava on this and other machines at LLNL.

Some of the benchmarks provided by the EPCC had bugs which were difficult to uncover. We were able to detect several bugs and send fixes to the EPCC for the Molecular Dynamics simulation. In particular, there was a problem with the way MPI Allreduce calls were being made, causing them to fail. As they were not failing consistently in the same place, it was not clear whether they were failing due to a system timeout, as has been experienced by both authors on the CASC Sun Cluster or for a host of other potential possibilities. This caused considerable difficulty as it was not clear whether the problem was with the mpiJava installation, how we were running the job, or a bug in the code.

The wrappers to run mpiJava were written to use mpirun or IBM's Load Leveler. These were not written with the the batch scheduling system, DPCS, in mind and we had to modify them (sometimes substantially) in order to get them to work. Also the first version of mpiJava that we tried would not compile properly on Blue due to a linking problem.

We asked LC to install more recent JVMs so that we could test some of the examples on ASCI Blue-Pacific, the Linux/Alpha cluster and the Compaq Tera 2000 system. This took considerable time as the system administrators on the larger systems are rather busy. It was unfortunate that root access was necessary to install JVMs for the TC2K machine and on ASCI Blue-Pacific; it would have been useful to try out some of the newer compilers to see what performance improvements have been made.

It should be noted that none of the benchmark applications would be considered a true large-scale application at the laboratory. The benchmark codes were not particularly memory intensive and perhaps were not the ideal codes to test for use at LLNL. It would be interesting to see how a "real" large-scale application code ported to Java would perform on LLNL systems. Due to the time constraints of this study, we could not investigate this further.

6 Conclusions

A general perception exists amongst the computational science community that Java is too slow for large-scale scientific applications. A number of studies have begun to counter this notion. Our analysis of Java on high-performance ASCI-class systems in this study seems to indicate that Java could be suitable for the development of large-scale scientific applications.

We focused our attention on three particular areas of Java performance; serial performance of Java relative to C, the performance of the mpiJava interface relative to currently-used MPI implementations, and memory overhead of Java relative to C. Serial and parallel tests were performed on different systems at LLNL.

The serial performance of Java applications were 1.1 to 2.0 times slower than the performance of comparable C benchmarks on a Sun. On IBM ASCI Blue-Pacific, Java applications were 2.5 to 3.3 times slower. We tested only standard, non-proprietary, JVMs so this result could change as faster JVMs are continually being developed by a variety of vendors. The performance of the mpiJava interface was essentially identical to the MPI implementations that are currently used for scientific applications at LLNL, suggesting that use of Java would not invoke any additional overhead in communications costs for parallel applications. Indeed, we tested Java applications that use mpiJava on up to 64 processors of Blue-Pacific and found reasonable scaling, although we could not make a direct comparison between Java and C for parallel performance because we did not have a parallel version of the C benchmark. We found that Java applications use more memory than C, particularly on Blue-Pacific. The overhead is primarily from the JVM and amounted to between 13 MB and 42 MB on the Sun. For larger-scale applications, it does not appear that this overhead would be a severe detriment.

One obvious advantage of using Java rather than C or Fortran is that it is an object-oriented language. A performance study comparing Java to C++, the most frequently used language for object-oriented programming at LLNL, would be an interesting topic for future work. It seems that using Java in the same way that C++ is being used at the laboratory is probably the most natural fit. It would be interesting to see a truly large-scale application written in Java to compare to it's C++/C or C++/Fortran counterpart.

Aside from purely performance related issues, we found a number of advantages with Java over traditional C and Fortran. Java code is significantly easier to debug and link together. A number of nice development environment's (e.g., IDEs) exist for Java which made programming quite easy. Executables run unchanged on multiple platforms [2]. Automatic garbage collection makes memory management extremely easy. Finally, because Java is used much more frequently today in the software development community than C or Fortran, there are a host of useful resources (e.g., development environments, re-usable code, books, tools) that could be leveraged for development of scientific software at LLNL.

Our experiences with Java in this study convinced us that the performance loss and memory overhead we observed is not detrimental enough to outweigh the advantages discussed above. The performance of Java, both in speed and memory, seems to rest squarely on the JVM. We found very different results for the JVMs implemented on the Sun and IBM. This work only investigated standard freely-available JVMs that are 1-2 years old. Future work should address performance of more recent proprietary JVMs on even larger problems.

7 Acknowledgments

We acknowledge the assistance of Bryan Carpenter at Indiana University for his assistance with installation of mpiJava. The benchmark application codes used in the study were supplied by the Edinburgh Parallel Computing Centre (EPCC). We also acknowledge the system administrators at LLNL for installing current versions of Java on the large-scale parallel systems used in the study. This work was funded by the CAO Tech Base Initiative in the computation directorate at LLNL in the initial year of the program. This program encourages extension of software technology and we support its continuance and growth in the Computing Applications and Research (CAR) department of the Computations directorate.

[2] We were able to compile a Java byte-code on a 200MHz Pentium at home and run it unchanged on IBM ASCI Blue-Pacific

References

[1] Java Grande Forum. See `http://www.javagrande.org`.

[2] B. Carpenter, G. Fox, S.-H. Ko, and S. Lim. *mpiJava 1.2: API Specification*, October 1999. See `http://aspen.csit.fsu.edu/pss/reports/mpiJava-spec/mpiJava-spec.ps`.

[3] Java Grande/ISCOPE. *Proceedings of the ACM 2001 Java Grande/ISCOPE Conference*, June 2001. Stanford University, Palo Alto, CA.

[4] Edinburgh Parallel Computing Centre (EPCC), University of Edinburgh. See `http://www.epcc.ed.ac.uk`.

[5] *Java Grande Benchmark Suite.* Edinburgh Parallel Computing Centre, University of Edinburgh. See `http://www.epcc.ed.ac.uk/javagrande`.

[6] Manpage for the jr utility. ASCI Blue-Pacific, LLNL.

[7] B. Carpenter. *Documentation included with mpiJava.* See `http://aspen.ucs.indiana.edu/pss/HPJava/mpiJava.html`.